

```

### Ode2c.R -- Translate an Odein language source file into c
###
### 1. Read in the file, return the ode source as a vector of strings.
### 2. Strip out all the comments. If the file starts with a comment,
###    save the entire block as documentation for the model. This will
###    become a ".Rd" file, along with the names of the parameters that
###    may be modified, and the names of variables whose values can be
###    requested.
### 3. Break the original vector of strings into a list of lists of
###    statements, one for each block.
### 4. Parse each block separately, that is, there is a separate function
###    for parsing each block.
###
### These are the main parsing and output routines. Utility routines are
### in Ode2c-utils.R.
###
### parse1(ode) -- This first pass breaks up ode (stripped of comments)
###               into a list of vectors of strings, one for each of the
###               blocks in the program.
###
###               The return value of parse1 is a list with named elements:
###               PARAMETERS, FUNCTIONS, STATE, DERIVS, EVENTS. Some of these
###               lists may be null.

```

```

parse1 <- function(ode)
{
  ## Extract the Initial documentation blocks:
  ## @Title:
  ## @Version:
  ## @Date:
  ## @Author:
  ## @License:
  ## @BEGIN Description ... @END Description
  ## @BEGIN DOC ... @END DOC
  ## @BEGIN Refs ... @END Refs
  ## Copy them into the documentation structure
  Doc <- vector("list",11)
  names(Doc) <- c("Title","Version","Date","Author","Description","Doc",
                 "Refs","ParmDocs","StateDocs","ExtraDocs","License")
  Doc>Title <- getDocTag(ode,"Title")
  Doc$Version <- getDocTag(ode,"Version")
  Doc$Date <- getDocTag(ode,"Date")
  Doc$Author <- getDocTag(ode,"Author")
  Doc$License <- getDocTag(ode,"License")
  Doc$Description <- getLongDoc(ode,"Description")
  Doc$Doc <- getLongDoc(ode,"DOC")
  Doc$Refs <- getLongDoc(ode,"Refs")
  ## Drop the blank lines
  ode <- ode[ode != ""]
  ## Find where each block starts and ends: that is, find all the
  ## "BEGIN"s and matching "ENDS"
  tmp <- list(PARAMETERS=character(0),
              FUNCTIONS=character(0),
              CONTINUOUS=character(0),
              VARIABLES=character(0),

```

```

JUMPS=character(0)
## Make a list of the locations:
blockstarts <- numeric(length(tmp))
names(blockstarts) <- names(tmp)
## find the BEGIN lines
for (f in names(tmp)) {
  pattern <- paste("^[:space:]*BEGIN[:space:]*",f,sep="")
  thisstart <- grep(pattern,ode)
  if (length(thisstart) > 0) {
    if (length(thisstart) > 1) {
      stop(paste("Multiple definitions of",f,"block\n"))
    } else {
      blockstarts[f] <- thisstart
    }
  } else blockstarts[f] <- NA
}
## reorder blockstarts. This drops missing blocks.
blockstarts <- sort(blockstarts)
## The block END lines are redundant. Each block ends on the line before the next one
## begins, or the end of the file, for the last one.
blockends <- c(blockstarts[-1]-1,length(ode))
## assign the names
names(blockends) <- names(blockstarts)
## Modify each blockend so it points to "^[[:space:]]*END"
for (f in names(blockstarts)) {
  for (i in blockends[f]:blockstarts[f]) {
    if (length(grep("^[[:space:]]*END",ode[i])) > 0) {
      blockends[f] <- i
      break
    }
  }
}
## Break up ode into its blocks: each one is parsed somewhat differently
for (f in names(blockstarts))
  tmp[[f]] <- ode[(blockstarts[f] + 1):(blockends[f] - 1)]
list(tmp, Doc)
}

### input p is a vector of strings of parameter definitions
### input SortEm is a logical; if TRUE, statements are sorted
### so that no variables appear on the right hand side of a statement
### before it appears on the left hand side: no variable is used before
### it is defined.
###
### Skeleton of steps to be carried out:
### 1) Split up and combine statements so that each statement uses one string
### 2) Extract any ARRAY declarations; throw an error if there is more than one
### 3) Parse any ARRAY declaration into a list containing array name, dimensions, and documentation
string
### 4) Variable names may be subscripted like [xx,yy,...], where xx, yy, etc are numeric constants ONLY
### 5) All subscripted variable names must be in an ARRAY declaration, and subscripts must be within
the
### specified dimensions (arrays begin subscripting at 1).
### 6) statements like var1 = var2 need to be checked:
### - var1 and var2 are both arrays
### dim(var1) == dim(var2)

```

```

### 7) replace references to var[x,y,...] with var_x_y...
### 8) replace var1 = var2 where both var1 and var2 are ARRAYs, with
###     var1_1 = var2_1
###     var1_2 = var2_2
###     ...
###     for all dimensions
### 9) expand all brace initializations
### 10) Parse each statement in the remaining statements; they are all in the form var1 = expression
### 11) 'expression' may be:
###     - a numeric constant
###     - an algebraic expression using allowed function names and other parameters
### 12) Extract dependencies from parse trees.
### 13) Check for circularities.
### 14) (optionally) Sort statements so that no variable is used before it is defined.
### 15) at top of Docstrings, add the Array dimensions and doc strings

### Return value is a list with elements:
### Arraydec (may be NULL)
###Parms = Vector of values for all parameters. Parameters with computed
###      values are assigned NAs. names(Parms) gives all parameter names.
###ConstParms = names of parameters initialized as numeric constants.
###      Only these parameters may be assigned new values at the
###      beginning of a simulation run.
###CompParms = text version of n R function for computing all parameter
###      values.
###Depends = dependency structure
###Docs = Docstrings
parse.Parameters <- function(p,SortEm)
{
  ## Split and combine statements so each takes up exactly one string.
  ## First, make one long string
  tmp <- paste(p, collapse=" ")
  ## Insert newlines before the beginning of each line.
  ## Just break the lines at ','
#  tmp2 <
 gsub("(([[:alpha:]]|[:alnum:]]*\\[[:digit:]]+[:space:]*)*([[:space:]]*,[[:space:]]*[:digit:]+)*\\)?([[:space:]]*[=])|ARRAY)", "\n\\1",tmp)
  tmp3 <- strsplit(tmp, ",")[[1]]
  ## drop blank lines
  tmp3 <- tmp3[tmp3 != ""]
  ## delete semicolon and surrounding spaces
  tmp3 <- sub("[:space:]*;[:space:]*","",tmp3)
  ## delete trailing spaces
  tmp3 <- sub("[:space:]*$","",tmp3)
  ## delete leading spaces
  tmp3 <- sub("^[:space:]*","",tmp3)
  ## separate any ARRAY declaration
  Arrayindx <- grep("^[[:space:]]*ARRAY", tmp3)
  if (length(Arrayindx) > 1) {
    writeLines(strwrap(tmp3[Arrayindx]))
    stop("Only one ARRAY declaration allowed in PARAMETERS")
  }
  if (length(Arrayindx) > 0) {
    Arraydec <- doArray(tmp3[Arrayindx])
    tmp3 <- tmp3[-Arrayindx]
  } else {

```

```

Arraydec <- NULL
}
## Extract the documentation strings and remove them from the input
Docstr <- getDocStr(tmp3)
tmp3 <- cleanDocStr(tmp3)
## Check references to array elements. Make sure all array references are within bounds

### tmp3 is now a vector of strings of the form var = expression
## Expand the brace initializations
tmp3 <-
  unlist(lapply(tmp3, function(z) {
    indx <- grep("^[[:alpha:]]|[[:alnum:]]*[[:space:]]*=[[:space:]]*\\"",z)
    if (length(indx) == 0) {
      z
    } else {
      ## get the lhs, and make sure it has been declared
      nm <- sub("([[:alpha:]]|[[:alnum:]]*)[[:space:]]*=".,"\\1",z)
      if (!(nm %in% names(Arraydec))) {
        writeLines(c("PARAMETERS:",z))
        stop("array not declared")
      }
      ## get the dimensions
      dim <- Arraydec[[nm]]$dim
      nelts <- prod(dim)
      ## get the rhs
      rhs <- strsplit(z,"[[:space:]]*=[[:space:]]*")[[1]][2]
      ## split it into elements
      elts <- strsplit(rhs, "[{}.]")[[1]]
      elts <- elts[-grep("^[[[:space:]]*\"",elts)]
      ## make sure we have enough initializers
      if (length(elts) != nelts) {
        writeLines(c("PARAMETERS:",z))
        stop("wrong number of elements in initializer")
      }
      ## strip off leading whitespace
      elts <- sub("^[[[:space:]]*","",elts)
      ## strip off trailing whitespace
      elts <- sub("[[[:space:]]*\"","",elts)
      zz <- do.call("expand.grid",lapply(dim, seq))
      append <- apply(data.matrix(zz), 1, function(z) paste("_",paste(z,collapse="_"),sep=""))
      paste(nm,append," =",elts,sep="")
    }}))

### run through the vector, checking each reference of the form
### [[[:alpha:]]|[[:alnum:]]*\\"[[[:digit:]]+([[:digit:]]+)*\\"]
### for indices in bounds, and declaration in ARRAY. if OK, replace with internal form
tmp3 <-
  sapply(tmp3, function(x) {
    if (length(grep("([[:alpha:]]|[[:alnum:]]*\\"[[[:digit:]]+([[:digit:]]+)*\\]",x)) == 0) {
      x
    } else {
      ## get all the subscripted names, and split them into name dim pairs
      y <- sapply(strsplit(gsub("([[:alpha:]]|[[:alnum:]]*\\"[[[:digit:]]+([[:digit:]]+)*\\]","\\n\\1",x),"\\n\\1",y)[[1]],function(z) {
        sub("(\\]).*$","\\1",z)})
      nm <- sub("([[:alpha:]]|[[:alnum:]]*)\\[.*$","\\1",y)
    }
  })

```

```

nm <- nm[nm != ""]
if (!all(zzz <- nm %in% names(Arraydec))) {
  stop(paste("Variables", paste(nm[!zzz], collapse=", "), "used as arrays, but not declared"))
}
## if we get here, all names were declared, so check that the references are within bounds
## get the array references in a named list, as vectors
dims <- sub("[[:alpha:]_][[:alnum:]]*\\\[([[:digit:]]+([[:digit:]]+)*\\]", "c(\\1)", y)
dims <- dims[dims != ""]
dims <- lapply(dims, function(z) eval(parse(text=z)))
## for each reference, check that the indices are in bounds: right number of dimensions,
## and reference < max array extent
for (i in seq(along=dims)) {
  if (length(dims[[i]]) != length(Arraydec[[nm[i]]]$dim)) {
    writeLines(paste("PARAMETERS Line:\\n", x))
    stop(paste(nm[i], "is used with the wrong number of dimensions"))
  }
  if (any(dims[[i]] > Arraydec[[nm[i]]]$dim)) {
    writeLines(paste("PARAMETERS Line:\\n", x))
    stop(paste(nm[i], "dimension is out of bounds"))
  }
}
x <- gsub("\\]", "", x)
x <- gsub("[.]", "_", x)
x
})
## check references like var1 = var2 to make sure the two variables
## have the same extent,
## and expand them to var1_1_1 = var2_1_1, etc.
tmp3 <- unlist(lapply(tmp3, function(z) {
  ## is this of the form var1 = var2, where both var1 and var2 are in names(Arraydec)?
  indx <- grep("^[[[:alpha:]_][[:alnum:]]*[[[:space:]]*=[[:space:]]*[[[:alpha:]][[:alnum:]]*\"", z)
  if (length(indx) == 0) {
    z
  } else {
    ## Get the two names
    nms <- strsplit(z, "[[:space:]]*=[[:space:]]*")[[1]]
    ## are they both in names(Arraydec)?
    if (any(nms %in% names(Arraydec)) && !all(nms %in% names(Arraydec))) {
      writeLines(c("PARAMETERS:", z))
      stop("Both sides of assignment must be declared in ARRAY")
    } else {
      ## Do the two arrays have the same extent?
      if (any(Arraydec[[nms[1]]]$dim != Arraydec[[nms[2]]]$dim)) {
        writeLines(c("PARAMETERS:", z))
        stop("In this sort of assignment, the arrays must have the same extent")
      }
    }
    ## We're OK to expand
    dims <- Arraydec[[nms[1]]]$dim
    zz <- do.call("expand.grid", lapply(dims, seq))
    append <- apply(data.matrix(zz), 1, function(z) paste("_", paste(z, collapse="_"), sep=""))
    paste(nms[1], append, " = ", nms[2], append, sep="")
  }
}))

```

```

## Parse the text
tmp4 <- parse(text=tmp3)
## evaluate all the constants constructed using arithmetic expressions
for (i in 1:length(tmp4)) {
  if (length(tmp4[[i]][[3]]) > 0 && length(all.vars(tmp4[[i]][[3]])) == 0)
    tmp4[[i]][[3]] <- eval(tmp4[[i]][[3]])
}
## construct the dependencies
tmp5 <- makeDep(tmp4)
## Make sure all the variables on the rhs appear on the lhs
rhsvars <- sort(unique(unlist(sapply(tmp5,function(x)x[[2]]))))
## get the lhs's
lhs <- sapply(tmp5,function(x)x[[1]])
if (any(Bad <- !(rhsvars %in% lhs))) {
  writeLines(strwrap(paste(paste(rhsvars[Bad], collapse=", "),
    "appear on the right hand side of an expression in PARAMETERS,",
    "but not on the left\n")))
  stop("Likely typos in PARAMETER names or missing PARAMETER definitions.", call.=FALSE)
}
## Check that all array elements are initialized. That is, they must all appear on the
## LHS of a statement.
## Construct a vector of all legal array names, and make sure they all appear in lhs
if (!is.null(Arraydec)) {
  Arrayelts <- unlist(lapply(names(Arraydec), function(nm) {
    dims <- Arraydec[[nm]]$dim
    zz <- do.call("expand.grid",lapply(dims, seq))
    append <- apply(data.matrix(zz), 1, function(z) paste("_",paste(z,collapse="_"),sep=""))
    paste(nm,append,sep="")
  }))
  if (!all(whichfail <- Arrayelts %in% lhs)) {
    writeLines(paste(Arrayelts[!whichfail], collapse=", "))
    stop("Array elements undefined in PARAMETERS",call.=FALSE)
  }
}
## Make sure all functions are legal
legalfns <- rownames(legalfunctions[legalfunctions$Parameters,])
for (i in seq(along=tmp4)) {
  fns <- all.funcs(tmp4[[i]])
  if (length(fns) > 0) {
    ## They'd better all be in names(legalfunctions)
    if (!all(fns %in% legalfns)) {
      writeLines(c(paste(fns[!(fns %in% legalfns)], collapse=", "),
        "are not allowed functions in Parameters"))
      stop("Illegal function call in PARAMETERS", call.=FALSE)
    }
  }
}
## If we are sorting, then make sure everything is defined only once
## then construct a dependence list, check for circularities, and sort
if (SortEm) {
  ## any duplicates?
  if (any(duplicated(lhs)))
    stop("Error: Sorting requested but variable in PARAMETER block defined more than once",
      call.=FALSE)
}

```

```

## Finish making the dependence list, as defined above.
tmp6 <- sapply(tmp5,function(x)x[[2]])
names(tmp6) <- sapply(tmp5,function(x)x[[1]])
Depends <- tmp6
## Check for circularities
if (!depcheck(tmp6,NULL,NULL))
  stop("Error: Circularities in Parameter variable definitions.",
       call.=FALSE)
## sort the statements
indx <- orderStatements(tmp5)
tmp4 <- tmp4[indx]
## reconstruct tmp5
tmp5 <- makeDep(tmp4)
} else {
  ## Make sure they are already sorted
  ## let rhs[i] be the set of all variables that appear on the rhs
  ## of statements 1:i; let lhs[i] be the set of all variables that
  ## appear on the lhs of statements 1:i. rhs[i] must be a subset of
  ## lhs[i-1], and rhs[1] must be a constant.
  ##
  ## Generate the subsets:
  lhs <- rhs <- vector("list",length(tmp5))
  lhs[[1]] <- tmp5[[1]][[1]]
  rhs[[1]] <- tmp5[[1]][[2]]
  for (i in 2:length(tmp5)) {
    lhs[[i]] <- c(lhs[[i-1]],tmp5[[i]][[1]])
    rhs[[i]] <- c(rhs[[i-1]],tmp5[[i]][[2]])
    if (!all(rhs[[i]] %in% lhs[[i-1]]))
      stop(paste(as.character(tmp4[[i]]),
                 "in PARAMETERS contains undefined variables"))
  }
}
## Construct the return values
## First, the parameter vector
Parms <- sapply(tmp4, function(x)if(length(x[[3]]) == 1 && is.numeric(x[[3]])) x[[3]] else NA)
names(Parms) <- sapply(tmp5,function(x)x[[1]])
## Next, the names of parameters that are given as constants (these can be
## modified
LogicalConst <- sapply(tmp4,function(x)length(x[[3]]) == 1 && is.numeric(x[[3]]))
ConstParms <- sapply(tmp5[LogicalConst], function(x)x[[1]])
## Finally, the function to calculate the 'computed' parameters
if (any(!LogicalConst)) {
  compbody <- tmp4[!LogicalConst]
  ## Replace variable name with 'Pm["variable name"]' throughout compbody
  ## be careful not to replace functions!
  compbody <- lapply(compbody,function(x) {
    x[[1]] <- as.name("<-")
    Var2Array(x,all.vars(x),"Pm")
  })
  CompParms <-
  c("## Pm is the vector of constant parameters",
    "function(Pm) {",
    "  nparms <-",
    "  c(",
    paste("  ",
      strwrap(paste(paste("\\"",sort(setdiff(names(Parms),ConstParms)),

```

```

        "","","sep=""),
        collapse=", ")),
        sep=""),
      " "),
      " qparms <- numeric(length(nqparms))",
      " names(qparms) <- nqparms",
      " Pm <- c(Pm, qparms),

      paste(" ",as.character(compbbody),sep=""),
      " Pm",
      "}")
      CompParms <- sub("[[:space:]]*\"", "",CompParms)
}
## Partition Docstr into Docstr$ConstParms and Docstr$CompParms
xx <- Docstr

Docstr <- vector("list",2)
Docstr$ConstParms <- xx[intersect(ConstParms,names(xx))]
Docstr$CompParms <- xx[match(setdiff(names(xx),ConstParms),names(xx))]
## Return:
list(Arrays=Arraydec,Parms=Parms,ConstParms=ConstParms,
      CompParms=CompParms,Depends=Depends,Docs=Docstr)
}

### input p is a vector of strings of State variable initializations
### input ParmNames is a vector of names of parameters;
### input ParmArrays is the Arrays component of the output from
###     parse.Parameters()
### input SortEm is a logical - if TRUE, statements are sorted so that no
###     value is used before it is defined.
###
### A complexity here is that the contents of this block are divided
### into STATE variables which have initializations, and others
### (extra) variables, whose values are desired, but which are the
### results of other computations. The STATE variables are defined
### within BEGIN STATE ... and END
###
### Skeleton of steps to be carried out:
### 1) Split up and combine statements so that each statement uses one string
### 2) Extract any ARRAY declarations; throw an error if there is more than one
### 4) Parse any ARRAY declaration into a list containing array name,
###    dimensions, and documentation string
### 5) Parse the KEEP declarations into a list containing variable name
###    and documentation string.
### 6) Variable names may be subscripted like [xx,yy,...], where xx, yy, etc
###    are numeric constants ONLY
### 7) All subscripted variable names must be in an ARRAY declaration,
###    and subscripts must be within the
###    specified dimensions (arrays begin subscripting at 1).
### 8) statements like var1 = var2 need to be checked:
###    - var1 and var2 are both arrays
###    dim(var1) == dim(var2)
### 9) replace references to var[x,y,...] with var_x_y...
### 10) replace var1 = var2 where both var1 and var2 are ARRAYS, with
###     var1_1 = var2_1
###     var1_2 = var2_2

```

```

### ...
### for all dimensions
### 11) expand all brace initializations
### 12) Parse each statement in the remaining statements; they are all in the
###     form var1 = expression
### 13) 'expression' may be:
###     - a numeric constant
###     - an algebraic expression using allowed function names and other
###         parameters
### 14) Extract dependencies from parse trees.
### 15) Check for circularities.
### 16) (optionally) Sort statements so that no variable is used before
###     it is defined.
### 17) at top of Docstrings, add the Array dimensions and doc strings

### Return value is a list with elements:
### Arraydec (may be NULL)
### State = Vector of values for all state variables that require initialization.
###     Variables with computed
###         initial values. c(names(State), names(Keep)) gives all state variable names.
### Keep = variables declared as state variables, but which do not require initial
###         values (because they are algebraic functions of other variables).
### initState = text version of n R function for computing all parameter
###         values.
### Depends = dependency structure
### Docs = Docstrings

```

```

parse.State <- function(p,ParmNames,ParmArrays, SortEm)
{
  ## Array declarations come first, then the STATE block,
  ## then (finally) all the extra (or derived) variables
  ## We need to process these separately.
  ## Split out the STATE and Extra blocks
  Statestart <- grep("^[[:space:]]*BEGIN *STATE",p)
  Stateend <- grep("^[[:space:]]*END",p[Statestart:length(p)])

  ## Split and combine statements so each takes up exactly one string.
  ## First, make one long string
  tmp <- paste(p, collapse=" ")
  ## split at ;
  tmp3 <- strsplit(tmp,";")[1]
  ## drop blank lines
  tmp3 <- tmp3[tmp3 != ""]
  ## delete semicolon and surrounding spaces
  tmp3 <- sub("^[[:space:]]*;[[:space:]]*", "",tmp3)
  ## delete trailing spaces
  tmp3 <- sub("^[[:space:]]*\\$","",tmp3)
  ## separate any ARRAY declaration
  Arrayindx <- grep("^[[:space:]]*ARRAY", tmp3)
  if (length(Arrayindx) > 1) {
    writeLines(strwrap(tmp3[Arrayindx]))
    stop("Only one ARRAY declaration allowed in STATE",
         call.=FALSE)
  }
  if (length(Arrayindx) > 0) {

```

```

Arraydec <- doArray(tmp3[Arrayindx])
tmp3 <- tmp3[-Arrayindx]
} else {
  Arraydec <- NULL
}
## separate and process any KEEP declaration
Keepindx <- grep("^[[:space:]]*KEEP",tmp3)
if (length(Keepindx) > 1) {
  stop("Only one KEEP declaration allowed in STATE",
    call.=FALSE)
}
if (length(Keepindx) > 0) {
  Keepdec <- doKeep(tmp3[Keepindx])
  tmp3 <- tmp3[-Keepindx]
} else {
  Keepdec <- NULL
}
## Extract the documentation strings and remove them from the input
Docstr <- getDocStr(tmp3)
tmp3 <- cleanDocStr(tmp3)
## Check references to array elements. Make sure all array references are within bounds

### tmp3 is now a vector of strings of the form var = expression
## Expand the brace initializations
tmp3 <-
  unlist(lapply(tmp3, function(z) {
    indx <- grep("^[[:alpha:]]_[[[:alnum:]]]*[[[:space:]]*=[[:space:]]*\\" ,z)
    if (length(indx) == 0) {
      z
    } else {
      ## get the lhs, and make sure it has been declared
      nm <- sub("([[:alpha:]]_)[[:alnum:]*][[:space:]]*=".,"\\1",z)
      if (!(nm %in% names(Arraydec))) {
        writeLines(c("STATE:",z))
        stop("array not declared")
      }
      ## get the dimensions
      dim <- Arraydec[[nm]]$dim
      nelts <- prod(dim)
      ## get the rhs
      rhs <- strsplit(z,"[[[:space:]]*=[[:space:]]*")[[1]][2]
      ## split it into elements
      elts <- strsplit(rhs, "[{}]")[1]
      elts <- elts[-grep("^[[:space:]]*\"",elts)]
      ## make sure we have enough initializers
      if (length(elts) != nelts) {
        writeLines(c("STATE:",z))
        stop("wrong number of elements in initializer")
      }
      ## strip off leading whitespace
      elts <- sub("^[[:space:]]*", "",elts)
      ## strip off trailing whitespace
      elts <- sub("[[:space:]]*$","",elts)
      zz <- do.call("expand.grid",lapply(dim, seq))
      append <- apply(data.matrix(zz), 1, function(z) paste("_",paste(z,collapse="_"),sep=""))
      paste(nm,append," =",elts,sep="")
    }
  })
}

```

```

    }))

### run through the vector, checking each reference of the form
### [[:alpha:]_][[:alnum:]_*\\[[[:digit:]]+([[:digit:]]+)*\\]]
### for indices in bounds, and declaration in ARRAY. if OK, replace with internal form
### array references may be STATE or PARAMETER arrays
tmp3 <-
  sapply(tmp3, function(x) {
    if (length(grep("[[:alpha:]_][[:alnum:]_*\\[[[:digit:]]+([[:digit:]]+)*\\]",x)) == 0) {
      x
    } else {
      ## get all the subscripted names, and split them into name dim pairs
      y <- sapply(strsplit(gsub("[[:alpha:]_][[:alnum:]_*\\[[[:digit:]]+([[:digit:]]+)*\\]]",
                            "\n\\1",x),"\n")[[1]],function(z) {
        sub("(\\.).*$","\\1",z)})
      nm <- sub("[[:alpha:]_][[:alnum:]_*\\[.*$","\\1",y)
      nm <- nm[nm !=""]
      if (!all(zzz <- nm %in% c(names(Arraydec),names(ParmArrays)))) {
        stop(paste("Variables", paste(nm[!zzz],collapse=", "),"used as arrays, but not declared"))
      }
      ## if we get here, all names were declared, so check that the references are within bounds
      ## get the array references in a named list, as vectors
      dims <- sub("[[:alpha:]_][[:alnum:]_*\\[[[:digit:]]+([[:digit:]]+)*\\]]","c(\\1)",y)
      dims <- dims[dims !=""]
      dims <- lapply(dims, function(z) eval(parse(text=z)))
      ## for each reference, check that the indices are in bounds: right number of dimensions,
      ## and reference < max array extent
      for (i in seq(along=dims)) {
        if (length(dims[[i]]) != length(c(Arraydec,ParmArrays)[[nm[i]]]$dim)) {
          writeLines(paste("STATE Line:\\n",x))
          stop(paste(nm[i],"is used with the wrong number of dimensions"))
        }
        if (any(dims[[i]] > c(Arraydec, ParmArrays)[[nm[i]]]$dim)) {
          writeLines(paste("STATE Line:\\n",x))
          stop(paste(nm[i],"dimension is out of bounds"))
        }
      }
      x <- gsub("\\]"," ",x)
      x <- gsub("[.,]","_",x)
      x
    }
  })

## check references like var1 = var2 to make sure the two variables
## have the same extent,
## and expand them to var1_1_1 = var2_1_1, etc.
tmp3 <- unlist(lapply(tmp3, function(z) {
  ## is this of the form var1 = var2, where both var1 and var2 are in
  ## names(c(Arraydec),ParmArrays)?
  indx <- grep("^[[:alpha:]_][[:alnum:]_*[[[:space:]]*[[[:space:]]*[[[:alpha:]]][[:alnum:]]*\\$\",z)
  if (length(indx) == 0) {
    z
  } else {
    ## Get the two names
    nms <- strsplit(z, "[[:space:]]*=[[:space:]]*")[[1]]
    ## are they both in names(Arraydec)?
  }
}))
```

```

if (any(nms %in% c(names(Arraydec),names(ParmArrays)))&&!all(nms %in%
c(names(Arraydec),names(ParmArrays)))) {
  writeLines(c("STATE:",z))
  stop("Both sides of assignment must be declared in ARRAY")
} else {
  ## Do the two arrays have the same extent?
  if (any(c(Arraydec,ParmArrays)[[nms[1]]]$dim != c(Arraydec,ParmArrays)[[nms[2]]]$dim)) {
    writeLines(c("STATE:",z))
    stop("In this sort of assignment, the arrays must have the same extent")
  }
}
## We're OK to expand
dims <- c(Arraydec,ParmArrays)[[nms[1]]]$dim
zz <- do.call("expand.grid",lapply(dims, seq))
append <- apply(data.matrix(zz), 1, function(z) paste("_",paste(z,collapse="_"),sep=""))
paste(nms[1],append," =",nms[2],append,sep="")
}
)))

## Parse the text
tmp4 <- parse(text=tmp3)
## evaluate all the constants constructed using arithmetic expressions
for (i in 1:length(tmp4)) {
  if (length(tmp4[[i]][[3]]) > 0 && length(all.vars(tmp4[[i]][[3]])) == 0)
    tmp4[[i]][[3]] <- eval(tmp4[[i]][[3]])
}
## construct the dependencies
tmp5 <- makeDep(tmp4)
## Make sure all the variables on the rhs appear on the lhs
rhsvars <- sort(unique(unlist(sapply(tmp5,function(x)x[[2]]))))
## get the lhs's
lhs <- sapply(tmp5,function(x)x[[1]])
if (any(Bad <- !(rhsvars %in% c(lhs,ParmNames)))) {
  browser()
  writeLines(strwrap(paste(paste(rhsvars[Bad], collapse=", "),
    "appear on the right hand side of an expression in STATES,",",
    "but not on the left or in the PARAMETERS\n")))
  stop("Likely typos in STATE definitions.",",
    call.=FALSE)
}
## Check that all array elements are initialized. That is, they must all appear on the
## LHS of a statement.
## Construct a vector of all legal array names, and make sure they all appear in lhs
if (!is.null(Arraydec)) {
  Arrayelts <- unlist(lapply(names(Arraydec), function(nm) {
    dims <- Arraydec[[nm]]$dim
    zz <- do.call("expand.grid",lapply(dims, seq))
    append <- apply(data.matrix(zz), 1, function(z) paste("_",paste(z,collapse="_"),sep=""))
    paste(nm,append,sep="")
  }))
  if (!all(whichfail <- Arrayelts %in% lhs)) {
    writeLines(paste(Arrayelts[!whichfail], collapse=", "))
    stop("Array elements undefined in STATE",call.=FALSE)
  }
}
## Make sure all functions are legal

```

```

legalfns <- rownames(legalfunctions[legalfunctions$State,])
for (i in seq(along=tmp4)) {
  fns <- all.funcs(tmp4[[i]])
  if (length(fns) > 0) {
    ## They'd better all be in names(legalfunctions)
    if (!all(fns %in% legalfns)) {
      writeLines(c(paste(fns[!(fns %in% legalfns)], collapse=", "),
                  "are not allowed functions in State"))
      stop("Illegal function call in STATE", call.=FALSE)
    }
  }
}
## If we are sorting, then make sure everything is defined only once
## then construct a dependence list, check for circularities, and sort
if (SortEm) {
  ## any duplicates?
  if (any(duplicated(lhs))) {
    stop("Error: Sorting requested but variable in STATE block defined more than once",
         call.=FALSE)
  }
  ## Finish making the dependence list, as defined above.
  tmp6 <- sapply(tmp5,function(x)x[[2]][!(x[[2]] %in% ParmNames)])
  names(tmp6) <- sapply(tmp5,function(x)x[[1]])
  ## Check for circularities
  if (!depcheck(tmp6,NULL,NULL))
    stop("Error: Circularities in State variable definitions.",
         call.=FALSE)
  ## sort the statements
  indx <- orderStatements(tmp5,Defined=ParmNames)
  tmp4 <- tmp4[indx]
  ## reconstruct tmp5
  tmp5 <- makeDep(tmp4)
} else {
  ## Make sure they are already sorted
  ## let rhs[i] be the set of all variables that appear on the rhs
  ## of statements 1:i; let lhs[i] be the set of all variables that
  ## appear on the lhs of statements 1:i. rhs[i] must be a subset of
  ## lhs[i-1], and rhs[1] must be a constant.
  ##
  ## Generate the subsets:
  lhs <- rhs <- vector("list",length(tmp5))
  lhs[[1]] <- tmp5[[1]][[1]]
  rhs[[1]] <- tmp5[[1]][[2]]
  for (i in 2:length(tmp5)) {
    lhs[[i]] <- c(lhs[[i-1]],tmp5[[i]][[1]])
    rhs[[i]] <- c(rhs[[i-1]],tmp5[[i]][[2]])
    if (!all(rhs[[i]] %in% lhs[[i-1]]))
      stop(paste(as.character(tmp4[[i]]),
                 "in STATES contains undefined variables"))
  }
}
## Construct the return values
## First, the parameter vector
Parms <- sapply(tmp4, function(x)if(length(x[[3]]) == 1 && is.numeric(x[[3]])) x[[3]] else NA)
names(Parms) <- sapply(tmp5,function(x)x[[1]])
## Finally, the function to calculate the 'computed' parameters
LogicalConst <- sapply(tmp4,function(x)length(x[[3]]) == 1 && is.numeric(x[[3]]))

```

```

if (any(!LogicalConst)) {
  compbody <- tmp4[!LogicalConst]
  ## Replace variable name with 'Pm["variable name"]' throughout compbody
  ## be careful not to replace functions!
  compbody <- lapply(compbody,function(x) {
    x[[1]] <- as.name("<-")
    xx <- Var2Array(x,ParmNames,"Pm")
    Var2Array(xx, names(Parms),"y")
  })
  initState <-
  c("## Pm here is the complete vector of state variables",
    "function(Pm, ystart=NULL)", "{",
    "nmy <- c(",
    paste(" ",strwrap(paste(paste("\\"",names(Parms),"\\"",sep=""),collapse=",")),sep=""),
    ")",
    " y <- numeric(length(nmy))",
    " names(y) <- nmy",
    " if(is.null(ystart)) {",
    paste(" y[\"",names(Parms[!is.na(Parms)]),"\"] <- ",Parms[!is.na(Parms)],sep=""),
    paste(" ",as.character(compbody),sep=""),
    " } else {",
    "   if (length(y) != length(ystart) || (length(names(ystart)) > 0 &&,
    "     (any(!(names(y) %in% names(ystart)))) ||",
    "     any(!(names(ystart) %in% names(y))))) {",
    "       stop(\"ystart has wrong length or wrong names\")",
    "     } else y <- ystart[names(y)]",
    "   }",
    " y",
    "}")
  }
  ## Return:
  list(Arrays=Arraydec, State=Parms, Keep=Keepdec, initState)initState, Docs=Docstr)
}

```

```

### parse.Cont: parse the CONTINUOUS block. The output here will be the parsed text,
### documentation strings, and the names of the variables with derivatives, and
### variables without derivatives. Some tests:
### 1) every dervlhs should be in StateNames
### 2)
parse.Cont <- function(p,ParmNames,StateNames,SortEm)
{
  ## Combine statements so each takes up exactly one string.
  tmp <- paste(p, collapse=" ")
  ## tmp2 <- gsub("[a-zA-Z][a-zA-Z0-9._]*[[space:]]*=", "\n\\1",tmp)
  ## split at ',', before and after '{', and before and after '}'
  tmp2 <- gsub(";", "\n",gsub("{|}", "\n\\1\n",tmp))
  tmp3 <- strsplit(tmp2,"\n")[1]
  ## drop blank lines
  tmp3 <- tmp3[tmp3 != ""]
  ## delete trailing spaces
  tmp3 <- sub("[[:space:]]*$","",tmp3)
  ## Extract the documentation strings and remove them from the input
  Docstr <- getDocStr(tmp3)
  tmp3 <- cleanDocStr(tmp3)
  ## Replace var' with var_dot.
  tmp3 <- gsub("[a-zA-Z][a-zA-Z0-9._]*","", "\\1_dot",tmp3)

```

```

## Expand the FOR loops
tmp3 <- expandFor(tmp3)
## Evaluate all the subscripts so that they are simply numeric
## and replace the '[...]' notation with '_...._'
tmp3 <- evalSubs(tmp3)
## Parse the text
tmp4 <- parse(text=tmp3)
## evaluate all the constants constructed using arithmetic expressions
for (i in 1:length(tmp4)) {
  if (length(tmp4[[i]][[3]]) > 0 && length(all.vars(tmp4[[i]][[3]])) == 0)
    tmp4[[i]][[3]] <- eval(tmp4[[i]][[3]])
}
## construct the dependencies
tmp5 <- makeDep(tmp4)
## rhsvars is all the variables that appear on the rhs of any statement
## in the CONTINUOUS block.
rhsvars <- sort(unique(unlist(sapply(tmp5,function(x)x[[2]]))))
## get the lhs's
lhs <- sapply(tmp5,function(x)x[[1]])
## Make sure all the variables on the rhs appear on the lhs, or is a PARAMETER
## or STATE variable. The point is to make sure no variable is used without it having
## a defined value.
if (any(Bad <- !(rhsvars %in% c(lhs,ParmNames,StateNames)))) {
  writeLines(strwrap(paste(paste(rhsvars[Bad], collapse=", "),
                           "appear on the right hand side of an expression in CONTINUOUS,",
                           "but not on the left or in the PARAMETERS or STATES.",
                           "These variables are undefined.")))
  stop("Likely typos in CONTINUOUS definitions.",
       call.=FALSE)
}
## Extract names with derivatives into derivlhs

derivlhs <- gsub("[a-z][a-zA-Z0-9.]*)_dot","\\1",lhs[grep("_dot$",lhs)])
## Make sure these are all state variables
if (length(xx <- setdiff(derivlhs, StateNames)) > 0) {
  writeLines(strwrap(paste(xx, collapse=", "),
                     "are not STATE variables, but have derivatives.")))
  stop("Likely typos in derivative definitions.",
       call.=FALSE)
}
## Get the lhs values that are not formed by "statevar". These are
## the "extra" variables.
Extras <- lhs[-grep("_dot$",lhs)]
## Split these into State variables and Local variables:
Locals <- setdiff(Extras, StateNames)
ExtraState <- intersect(Extras, StateNames)
## Finally, variables in StateNames, but in neither derivlhs nor ExtraState, are either constant
## (which is OK), or change only in Events.
ExtraExtras <- setdiff(StateNames, c(derivlhs,ExtraState))
## If we are sorting, then make sure everything is defined only once
## then construct a dependence list, check for circularities, and sort
if (SortEm) {
  ## any duplicates?
  if (any(duplicated(lhs)))
    stop("Error: Sorting requested but variable in STATE block defined more than once",
         call.=FALSE)
}

```

```

## Finish making the dependence list, as defined above.
tmp6 <- sapply(tmp5,function(x)x[[2]][!(x[[2]] %in% c(StateNames,ParmNames))])

names(tmp6) <- sapply(tmp5,function(x)x[[1]])
## Check for circularities
DependCheck <- depcheck(tmp6,NULL,NULL)
if (!DependCheck)
  stop("Error: Circularities in variable definitions.",
       call.=FALSE)
## sort the statements
indx <- orderStatements(tmp5,Defined=c(ParmNames,StateNames))
tmp4 <- tmp4[indx]
## reconstruct tmp5
tmp5 <- makeDep(tmp4)
} else {
  ## Make sure they are already sorted
  ## let rhs[i] be the set of all variables that appear on the rhs
  ## of statements 1:i; let lhs[i] be the set of all variables that
  ## appear on the lhs of statements 1:i.  rhs[i] must be a subset of
  ## lhs[i-1], and rhs[1] must be a constant.
  ##
  ## Generate the subsets:
  lhs <- rhs <- vector("list",length(tmp5))
  lhs[[1]] <- tmp5[[1]][[1]]
  rhs[[1]] <- tmp5[[1]][[2]]
  for (i in 2:length(tmp5)) {
    lhs[[i]] <- c(lhs[[i-1]],tmp5[[i]][[1]])
    rhs[[i]] <- c(rhs[[i-1]],tmp5[[i]][[2]])
    if (!all(rhs[[i]] %in% lhs[[i-1]]))
      stop(paste(as.character(tmp4[[i]]),
                 "in STATES contains undefined variables"))
  }
}
## Return:
list(list(Derivs=tmp4, Extras=Extras),Docstr)
}

```

```

#### Events are the mechanism for changing state variables at discrete times (in the future, they
#### may also be triggered at particular conditions of state variables). The JUMPS block specifies
#### a series of functions that describe changes to state or parameter values
parse.Jumps <- function(p, ParmNames, StateNames, SortEm) {
  list(p)
}

parse2 <- function(xx, sortDerivs){
  Doc <- xx[[2]]
  ode <- xx[[1]]
  parmsout <- parse.Parameters(ode$PARAMETERS,sortDerivs)
  Doc$ParmDocs <- parmsout[[2]]
  statesout <- parse.State(ode$STATE,names(parmsout[[1]][[1]]),sortDerivs)
  Doc$StateDocs <- statesout[[2]]
  derivsout <-
  parse.Cont(ode$CONTINUOUS,
             names(parmsout[[1]][[1]]),
             names(statesout[[1]][[1]]),
             sortDerivs)

```

```

Doc$ExtraDocs <- derivsout[[2]]
Events <- parse.Jumps(ode$JUMPS,
  names(parmsout[[1]][[1]]),
  names(statesout[[1]][[1]]),
  sortDerivs)
c(parmsout[[1]],statesout[[1]],derivsout[[1]],Doc,Events=Events)
}
### p2DESC generates the description and NAMESPACE files
p2DESC <- function(x, bn) {
  x <- data.frame(Title=x>Title,
    Package=bn,
    Version=x$Version,
    Date=x>Date,
    Author=x$Author,
    Depends="deSolve, RDynamic",
    Description=paste(x>Description, collapse=" "),
    Maintainer=x$Author,
    License=x$License)
  descfile <- file.path(bn,"DESCRIPTION")
  write.dcf(x, file=descfile)
  nmfile <- file.path(bn,"NAMESPACE")
  writeLines(c(paste("export(`,bn,`)",sep=""),
    "importFrom(RDynamic, initparms, ConstructEvents)",
    paste("useDynlib(`,bn,`)",sep=""))
  ), con=nmfile)
}
### p2doc generates the documentation files
### p2R generates the R-language and data files
p2R <- function(x, bn) {
  ## generate bn.Parms vector
  dfile <- file.path(bn,"R",paste(bn,"Parms","R",sep="."))
  ## This is just the constant parameters. We sort them to make it
  ## easier to find particular values. This list needs to match exactly
  ## the first part of the list in bn.h
  constParms <- sort(x$ConstParms)
  Parmdocs <- x$ParmDocs$ConstParms[sort(constParms)]
  Parmdocs[is.na(Parmdocs)] <- ""
  Parmdocs <- ifelse(nchar(Parmdocs) > 0,paste(" ## ",Parmdocs,sep=""),"")
  nconst <- length(constParms)
  writeLines(c(
    paste(bn,"Parms <-",sep="."),
    " c(`",
    paste(" ",constParms[-nconst]," = ",x$Parms[constParms][-nconst],
      ",",
      Parmdocs[-nconst],
      sep=""),
    paste(" ",constParms[nconst]," = ",x$Parms[constParms][nconst],
      Parmdocs[nconst],sep=""),
    ")"
  ), con=dfile)
  ## Output a structure that contains parameter names,
  ## state variable names, and extra variable names.
  dfile <- file.path(bn,"R",paste(bn,"Names","R",sep="."))
}

```

```

pnms <- names(x$Parms)
writeLines(c(
    paste(bn,"Names <-,sep="."),
    " list( Parameters=c(",
    paste(pnms, collapse=","),
    "),",
    " States=c(",
    paste(x$State,collapse=","),
    "),",
    " Extras=c(",
    paste(x$Extras,collapse=","),
    "))"
), con=dfile)

## Initialization for parms and state:
initPfile <- file.path(bn,"R","ComputeParms.R")
writeLines(c("ComputeParms <",
            x$CompParms),
            con=initPfile)

initSfile <- file.path(bn,"R","Inity.R")
writeLines(c("Inity <",
            x$initState), con=initSfile)

## update and writeout the 'runmodel' file.
## This includes the actual runmodel file,
## event definitions,
## and the intial list of time events.
## First, modify runmodTemplate:
## We need:
## @ModName@ -> bn
## @NSTATE@ <- number of derivatives
## @NEXTRA@
## @KnownEvents@ <- should be 0 - Q calls to ev$insertEvent()
## @DERIVS@
## @DLLNAME@
## @INITNAME@
## @ParmVector@
## @OUTNAMES@

runmod <- gsub("@ModName@",bn,runmodTemplate)
runmod <- gsub("@NSTATE@",length(x$State),runmod)
runmod <- gsub("@NEXTRA@",length(x$Extras),runmod)
runmod <- gsub("@DERIVS@",paste(bn,"derivs",sep=""),runmod)
runmod <- gsub("@DLLNAME@",bn,runmod)
runmod <- gsub("@INITNAME@",bn,runmod)
runmod <- gsub("@KnownEvents@","# KnownEvents",runmod)
runmod <- gsub("@ParmVector@",paste(bn,"Parms",sep=".") ,runmod)
runmod <- gsub("@OUTNAME@",paste(bn,"Names$Extras",sep=".") ,runmod)
runfile <- file.path(bn,"R","runmodel.R")
writeLines(runmod, con=runfile)

## Unload the dynamic library when we detach the library.
dfile <- file.path(bn,"R","zzz.R")
writeLines(c(".onUnload <- function(libpath) {",
            paste("library.dynam.unload(",bn,",libpath)",sep=""),
            "}") ,con=dfile)
}

### p2c generates the c-language files

```

```

p2c <- function(x.bn)
{
  ## .h first
  hfile <- file.path(bn,"src",paste(bn,"h",sep="."))
  cat(paste("/*Translated from file ",bn,".ode at ",
            date()," *\n",sep=""),file=hfile)
  ## Parameters
  cat("/* Parameters */",
      file=hfile, sep="\n",append=TRUE)
  cat(paste("#define N_PARMS",length(x$Parms)),
      file=hfile, sep="\n",append=TRUE)
  cat(paste("static double _RDy_",bn,"_parms[N_PARMS];",sep=""),
      file=hfile,sep="\n",append=TRUE)
  ## Need to do the constant parms first, then the computed parms.
  ## To make it easier to find values, lets sort the names in each
  ## sublist.
  ## The first part of this list needs to coincide exactly with
  ## the list in bn/data/bn.Parms.R, written in p2R, above.
  pnms <- names(x$Parms)
  constParms <- sort(x$ConstParms)
  compParms <- sort(setdiff(pnms, constParms))
  cat(paste("#define ",constParms," _RDy_",bn,"_parms[",
            seq(0,length(constParms)-1)],",
            sep=""),file=hfile,sep="\n",append=TRUE)
  cat("\n/* Computed Parameters */",
      file=hfile, sep="\n", append=TRUE)
  cat(paste("#define ",compParms," _RDy_",bn,"_parms[",
            seq(length(constParms), length(x$Parms)-1)],",
            sep=""),file=hfile,sep="\n",append=TRUE)

  ## State Variables
  cat("\n/* State Variable Definitions */",
      file=hfile, sep="\n", append=TRUE)
  cat(paste("#define ",names(x$State),
            " _RDy_",bn,"_state[",seq(0,length(x$State)-1)],",
            sep=""),file=hfile,sep="\n",append=TRUE)

  ## Derivative Defines
  cat("\n/* Derivative Definitions */",
      file = hfile, sep="\n",append=TRUE)
  cat(paste("#define ",names(x$State), " _dot_RDy_",bn,"_ydot[",
            seq(0,length(x$State)-1)],",sep=""),
      file=hfile,sep="\n",append=TRUE)

  ## 'Extra' variables: these are quantities that are defined algebraically.
  ## They may have intrinsic interest, or may just be intermediate values.
  cat("\n/* Extra variables, defined algebraically */",
      file=hfile, sep="\n", append=TRUE)
  if (length(x$Extras) > 0) {
    cat(paste("#define ",x$Extras,
              " _RDy_",bn,"_extras[",seq(0,length(x$Extras)-1)],",
              sep=""),file=hfile,sep="\n",append=TRUE)
  }

  ## Derivative and Event definitions

```

```

cfile <- file.path(bn,"src",paste(bn,"c",sep="."))
con <- file(description=cfile, open="at")
writeLines(c(paste("/*Translated from file ",bn,".ode at ",
date()," *\n",sep="")),
"#include <R.h>",
paste("#include \"",basename(hfile),"\"",sep=""),
"\n/* initializer */",
paste("void ",bn,"(void(* odeps)(int *, double *))",
sep=""),
"\"",
" int N=N_PARMS;",
paste(" odeps(&N, _RDy_, bn,\"_parms);",sep=""),
"\"",
"",
"\n/* Derivatives */",
paste("void ",bn,
"derivs(int *neq, double *t, double *",
paste("_RDy_",bn,"_state,", sep=""),
" double *",
paste("_RDy_",bn,"_ydot,",sep=""),
" double *",
paste("_RDy_",bn,"_extras,",sep=""),
" int *ip",
") {",
sep=""),
if (length(x$Extras) > 0) paste(" if (ip[0] < ",length(x$Extras),
") error(\"nout should be at least ",
length(x$Extras),"\"");sep="") else "" ,
paste(" ",as.character(x$Derivs),";",sep=""),
"}"),con=con)
close(con)
}

```

```

Ode2c <- function(s,sortDerivs=TRUE, Delete=TRUE)
{
  ## if 's' is a string, it points to a file to open
  ## if it is class 'RDynModel', it already has code in it, and
  ## we just want to retranslate and compile it.
  if (is.character(s))
  {
    nm <- s
    ode <- getODE(s)
    s <- vector("list",3)
    s$ode <- ode
    s$name <- nm
    s$date <- date()
    class(s) <- "RDynModel"
  }
  else
    if(class(s) == "RDynModel")
    {
      nm <- s$name
    }
  else stop("Argument must be a string or of class RDynModel")
  ## If there is already a package in this directory with name "nm", delete it
  ## then recreate it. If it exists, warn the user first! Four choices,

```

```

## 1) delete, 2) overwrite, 3) allow the user to rename the model (renames the
## original .ode file as well, 4) cancel.
if(Delete && file.exists(nm)) unlink(nm, recursive=TRUE)
if (file.exists(nm)) {
  mres <- menu(c("Delete package","Overwrite package","Rename new model",
    "Exit Ode2c immediately"),
    title=paste("The package",nm,"already exists; action?"))
  nm <- switch(mres, {
    unlink(nm, recursive=TRUE)
    silent.package.skeleton(nm)
    nm
  },{nm}, {
    cat(paste("Current name is:",nm,"New name: "))
    mres2 <- 2
    while (mres == 2) {
      newnm <- readline(prompt=paste("Current name is:",nm,"New name: "))
      if (file.exists(newnm))
        cat("\n!! A package already exists with that name. If you continue, it will be overwritten\n")
      mres2 <- menu(c("OK","Try Again","Exit Ode2c immediately"),
        title=paste("Use ",newnm,"?",sep=""))
    }
    if (mres == 3) stop("User requested bail-out!")
    if (!file.rename(paste(nm,"ode",sep="."),
      paste(newnm,"ode",sep="."))
      ) stop("File rename failed.")
    ## Now create the package
    silent.package.skeleton(newnm)
    newnm
  }, {
    stop("User requested bail-out!")
  }

  )
  if (mres == 3) s$name <- nm
}

} else {
  silent.package.skeleton(nm)
}
## Create the nm(inst subdirectory, and copy the source there
dir.create(file.path(nm,"inst"))
writeLines(s$ode,con=file.path(nm,"inst",paste(s$name,"ode",sep="")))
## Most of the rest of what we want to do is work with the strings
ode <- s$ode
## Break the code into its constituent blocks, after deleting all the
## comments
xx <- parse1(stripComments(ode))
# ode <- parse2(ode,sortDerivs)
## Parse the Blocks, sorting the DERIVS if required
Doc <- xx[[2]]
ode <- xx[[1]]
parmsout <- parse.Parameters(ode$PARAMETERS,sortDerivs)
Doc$ParmDocs <- parmsout[[6]]
statesout <-
  parse.State(ode$VARIABLES,names(parmsout[[2]]),
    parmsout[[1]], sortDerivs)
Doc$StateDocs <- statesout[[5]]
derivsout <- parse.Cont(ode$CONTINUOUS, names(parmsout[[2]]),

```

```
    names(statesout[[2]]),sortDerivs)
Doc$ExtraDocs <- derivsout[[2]]
ode <- c(parmsout[1:5],statesout[1:4],derivsout[[1]],Doc,Events=NULL)
p2DESC(ode, nm)
p2c(ode, nm)
p2R(ode, nm)
ode
}
```